2650 mini assembler simplifies programming

Here is a handy "real time" assembler program for small 2650 microcomputer systems. You can use it to load programs directly into memory in mnemonic assembly language — much faster, easier and more reliable than having to do all the detailed coding and displacement calculations yourself!

by JAMIESON ROWE

Programming a computer in machine language tends to be a very slow and tedious business. If you're doing it this way at the moment, you'll know what I mean. It can be challenging enough to work out the basic flow of a program — then you have to sit down and painstakingly slog through the coding, instruction by instruction.

But time and tedium aren't the only problems. When you try running such a program coded by hand, the odds are that you'll find quite a few "bugs" caused by coding errors and mistakes in working out relative addressing displacements.

People using larger computers generally don't need to worry about such problems, because they don't have to program in machine language. In fact many couldn't do so even if they wanted to (which is unlikely), either because they've never learned how or because the operating system on their computer has no provision for loading or running programs in this form!

The closest such folk ever need to come to machine language is assembly language programming, using easy-to-remember mnemonic symbols for the various instructions. An assembler program running in the computer itself is then used to translate this symbolic version of the program into machine language. The assembler takes over all the detailed coding, and works out all of those tedious displacements. Not only that, but it does them much faster and far more reliably than mere humans!

Assemblers for some microcomputer systems have been available for quite a while now, giving users of these systems most if not all of the advantages possessed by larger systems. For industrial and commercial users of the 2650 microprocessor, Signetics themselves provide a "cross assembler" — an assembler for 2650 code which itself runs on another machine.

```
*G1600
2650 LINE ASSEMBLER

0440.*THIS IS A DEMONSTRATION
0440.*
0440.*
0440.    ORG 500
0500.    DATA 5 14
0502.    LODI,R3 FF        SET UP R3 AS INDEX
0504.    LODA,R3 *+500     FETCH CHAR
0507.    COMI,R0 00        CHECK IF EOF (NUL)
0509.    BCTA,EQ @1        LEAVE IF FOUND
050C.    ZBSR *20          OTHERWISE GO PRINT
050E.    BCTR,UN 504       & CONTINUE
0510.@1  ZBSR *25          END: GIVE CRLF
0512.    ZBRR 22           & LEAVE--RETURN TO PIPBUG
0514.    ASCI "HELLO THERE!"
0520.    DATA 0
0521.    END

*G500
HELLO THERE!
```

Fig. 1: A demonstration of the mini assembler in action. As you can see, a program may be run immediately following assembly.

For smaller 2650 systems, more conventional "resident" assemblers have recently become available. A limited-facility "line" assembler called Prometheus was developed by the British Mullard company, and made available in a special ROM/RAM application card. However it was rather too expensive for hobby applications. Similarly an assembler was developed within the 2650 Users' Group in Sydney, but was memory-orientated and required quite a deal of RAM memory. Neither assembler was really well suited for small hobby systems.

Now for the good news. In this article, you will find details of a new 2650 assembler which I believe is almost ideal for small hobby systems. It occupies only 1300-odd bytes, so that it should fit into almost any 2650 system. Yet it will let you perform convenient and fast assembly of programs, from your terminal keyboard and in real time. You type in the mnemonics; it works out the code and plugs it into memory.

As you might expect, it is not a full-scale assembler like those you would find on large systems. It is basically a line assembler, which treats each instruction as a separate entity. But it does offer a very useful feature not found on many small line assemblers: limited forward referencing, which lets a branch instruction reference a memory location not yet known. This means that once you get used to its limitations, you can do almost as many things with this assembler as you can with its bigger brothers.

Incidentally I can't take much of the credit for this assembler. I haven't written it from scratch, but have developed it from a small assembler called PIPLA written by the software people at Signetics. I came across PIPLA last year when I toured the Signetics plant during my trip to California.

The people at Signetics told me they had written PIPLA to go into a special ROM device along with a modified and enhanced version of PIPBUG. When I showed interest in it, they let me have a copy along with a source listing.

I didn't have much of a chance to look closely at PIPLA during the trip but was able to do so when I came home. It didn't take long then to make a rather important discovery. Not unex-

pectedly, PIPLA used quite a few utility routines from the modified PIPBUG — but the modified PIPBUG was so different from the familiar old PIPBUG that the two were virtually incompatible.

Obviously PIPLA in its original form was not going to be all that much use to all those 2650 users who were already committed to the old PIPBUG. If it was to be of value to such people, someone was going to have to sit down and convert it to use the routines in "old PIPBUG" . . . .

Well, the rest is fairly obvious. The job took a while, as it had to be fitted in between more urgent things. There were a few complications, because some of the required routines in the modified PIPBUG were so different from those in the old PIPBUG that the routines in "old PIPBUG" could not easily be used at all. I had to add these to PIPLA itself, at the same time reducing the size of PIPLA wherever possible to minimise the increase in memory space.

Eventually I finished the basic conversion job, and after the inevitable debugging the modified PIPLA began running on my system with "old PIPBUG". But this wasn't quite the end of the story.

Once you got used to its limitations, it was a very handy piece of software. But there were a few mildly irritating little shortcomings. When you called it, it simply printed out a suggested initial "origin" or starting address for assembly. Wouldn't it be nicer if it announced itself with a suitable message?

Similarly, it lacked a facility for accepting numbers and other data constants, in hexadecimal. Wouldn't it be nice if it had a "DATA" directive like bigger assemblers?

To cut a long story short, these facilities were added and the result is presented here. Based on PIPLA but with quite a bit of modification and a couple of additional features, it is quite a capable little assembler. Certainly you should find it a big step forward in speed and convenience if you're still programming in machine language.

What will it do? Well, it will accept all of the standard 2650 instruction mnemonics — LODA, STRR, BCTA, BSTR and so on. It can also recognise all of the commonly used register/condition code mnemonics R0, R1, R2, R3, P, Z, N, LT, EQ, GT and UN. It will accept symbols for indirect and indexed addressing, up to 10 label symbols for forward referencing, four different pseudo-operation or assembler directives, and comments.

The input format required by the assembler for the symbolic source lines is:

LBL OPC R/C SYM OPND

where the symbols have the following meaning:

```
15CC F4 0C 18 02
15D0 45 1F 6D 04 2A 1F 17 5B F4 04 98 0F 3F 1A 95 CE
15E0 84 0D 3B 0F EF 04 29 9A 25 1B 71 3B F0 1B 1D 02
15F0 69 1A CD 0D 04 0D 0E 04 0E DA 02 D9 00 1F 00 A4
1600 20 07 14 CF 5A 40 5B 7B 3F 1A 55 C0 3B F0 0D 04
1610 0D 3B DC 0D 04 0E 3B D7 04 2E BB A0 3B D3 0C 1A
1620 02 E4 2A 18 69 E4 40 98 3C 0F 3A 02 A4 30 1E 02
1630 50 C3 E7 09 19 F9 D3 06 01 0F 7A 40 CC 04 0F C1
1640 0F 7A 41 CC 04 10 61 18 1A 0C 84 0F CF 7A 40 0E
1650 E4 0F CF 7A 41 0C 04 0D CC 84 0F 0C 04 0E CE E4
1660 0F 1B 56 07 02 20 CC 04 2A 3B BE 3F 17 7A CC 04
1670 11 60 9A 2B 44 0F 1C 00 22 F4 02 98 9C 3B AA 0F
1680 7A 02 C2 0F 3A 02 EF 04 29 9E 16 0E E2 18 FB CC
1690 84 0D 02 3F 15 F3 C2 1B 6A 15 D8 C0 C0 1A 95 CE
16A0 84 0D E4 10 9A 21 87 01 3F 17 6B 0F 7A 02 E4 40
16B0 99 0A 3F 17 7A 84 10 18 05 1F 02 50 3B DF 46 03
16C0 0C 84 0D 62 CC 84 0D 3B CB 0C 04 11 F4 01 1C 16
16D0 0E F4 02 1C 17 22 3B D1 0F 7A 02 E4 30 9A 1B 87
16E0 01 05 FF ED 37 CF 18 06 E5 04 1A 77 1B CC D1 D1
16F0 D1 D1 D1 6D 04 2A C9 FC 1B 5C E4 40 98 24 0F 3A
1700 02 A4 30 1E 02 50 C3 E7 09 19 F9 D3 0F 7A 40 C1
1710 0F 7A 41 C2 0C 04 0D CF 7A 40 0C 04 0E CF 7A 41
1720 1B B7 3F 1A 95 0C 04 11 F4 08 18 AD F4 04 98 24
1730 77 09 A6 01 A5 00 77 01 AE 04 0E AD 04 0D 75 08
1740 18 0D 85 01 9C 02 50 F6 C0 98 FA 46 7F 1B 05 04
1750 C0 42 98 F1 6E 04 2A 1B 0A 15 CC CD 84 0D 02 3F
1760 15 F3 C2 CE 84 0D 3B F8 1F 16 0E 04 20 FB 00 EF
1770 3A 02 18 7B EF 04 29 9A CC 17 06 FC A7 01 0F 3A
1780 02 EB F2 9A 0D E4 30 1A 09 CE 79 40 DA 70 87 01
1790 1B 07 04 20 CE 79 40 DA 7B CF 04 28 75 01 77 08
17A0 05 17 06 D4 CD 04 0F CE 04 10 07 FF 0F A4 0F 1C
17B0 02 50 EF 7A 3C 18 06 86 06 85 00 1B 67 E7 03 1A
17C0 6B 0F A4 0F C2 0F A4 0F 0F 04 28 75 08 17 00 2C
17D0 2B 2D 23 2A 52 30 20 20 00 F0 52 31 20 20 01 F0
17E0 52 32 20 20 02 F0 52 33 20 20 03 F0 50 20 20 20
17F0 01 F0 5A 20 20 20 00 F0 4E 20 20 20 02 F0 4C 54
1800 20 20 02 F0 45 51 20 20 00 F0 47 54 20 20 01 F0
1810 55 4E 20 20 03 F0 45 4E 44 20 00 80 4F 52 47 20
1820 00 81 41 53 43 49 00 82 4C 4F 44 5A 00 01 4C 4F
1830 44 49 04 02 4C 4F 44 52 08 04 4C 4F 44 41 0C 08
1840 53 54 52 5A C0 01 53 54 52 52 C8 04 53 54 52 41
1850 CC 08 49 4F 52 5A 60 01 49 4F 52 49 64 02 49 4F
1860 52 52 68 04 49 4F 52 41 6C 08 41 4E 44 5A 40 01
1870 41 4E 44 49 44 02 41 4E 44 52 48 04 41 4E 44 41
1880 4C 08 45 4F 52 5A 20 01 45 4F 52 49 24 02 45 4F
1890 52 52 28 04 45 4F 52 41 2C 08 42 43 54 52 18 04
18A0 42 43 54 41 1C 0C 42 43 46 52 98 04 42 43 46 41
18B0 9C 0C 43 4F 4D 5A E0 01 43 4F 4D 49 E4 02 43 4F
18C0 4D 52 E8 04 43 4F 4D 41 EC 08 41 44 44 5A 80 01
18D0 41 44 44 49 84 02 41 44 44 52 88 04 41 44 44 41
18E0 8C 08 53 55 42 5A A0 01 53 55 42 49 A4 02 53 55
18F0 42 52 A8 04 53 55 42 41 AC 08 52 45 54 43 14 01
1900 52 45 54 45 34 01 42 53 54 52 38 04 42 53 54 41
1910 3C 0C 42 53 46 52 B8 04 42 53 46 41 BC 0C 52 52
1920 52 20 50 01 52 52 4C 20 D0 01 43 50 53 55 74 12
1930 43 50 53 4C 75 12 50 50 53 55 76 12 50 50 53 4C
1940 77 12 42 52 4E 52 58 04 42 52 4E 41 5C 0C 42 49
1950 52 52 D8 04 42 49 52 41 DC 0C 42 44 52 52 F8 04
1960 42 44 52 41 FC 0C 42 53 4E 52 78 04 42 53 4E 41
1970 7C 0C 4E 4F 50 20 C0 11 48 41 4C 54 40 11 54 4D
1980 49 20 F4 02 57 52 54 44 F0 01 52 45 44 44 70 01
1990 57 52 54 43 B0 01 52 45 44 43 30 01 57 52 54 45
19A0 D4 02 52 45 44 45 54 02 5A 42 53 52 BB 10 5A 42
19B0 52 52 9B 10 54 50 53 55 B4 12 54 50 53 4C B5 12
19C0 4C 50 53 55 92 11 4C 50 53 4C 93 11 53 50 53 55
19D0 12 11 53 50 53 4C 13 11 42 53 58 41 BF 1C 42 58
19E0 41 20 9F 1C 44 41 52 20 94 01 4C 44 50 4C 10 1C
19F0 53 54 50 4C 11 1C 44 41 54 41 00 84 00 00 00 00
1A00 00 00

1A55 05 1A 06 6C 3F 00 A4 07 FF 0F A4
1A60 0D 18 04 BB A0 1B 77 05 04 06 40 17 32 36 35 30
1A70 20 4C 49 4E 45 20 41 53 53 45 4D 42 4C 45 52 0D
1A80 0A 0A 00 00 A4 30 1A 0A E4 0A 16 A4 07 1A 03 E4
1A90 10 16 1F 02 50 20 C1 C2 CC 04 12 08 FC 15 EF 04
1AA0 29 14 0F 7A 02 E4 20 98 02 DB 70 3B 57 D2 D2 D2
1AB0 D2 CE 04 28 46 F0 62 C2 D1 D1 D1 D1 45 F0 08 F2
1AC0 44 0F 61 C1 04 01 C8 D1 DB 54 0A 0D 5E 07 00 E7
1AD0 3C 1C 00 1D 3F 02 86 E4 7F 98 0A 03 18 71 0F 5A
1AE0 02 BB A0 1B 6A 05 03 ED 7A C9 18 09 F9 79 CF 7A
1AF0 02 BB A0 DB 5A CF 04 29 CD 04 2A 07 00 9B A5
```

Fig. 2: A complete hex listing of the assembler. The gap from 1A02 to 1A54 is occupied by the input and labels buffers.

LBL is an optional label; if present it must be one of the labels used in the operand field of a previous instruction, for forward referencing.

OPC is the instruction or pseudo-operation mnemonic; the standard 2650 mnemonics are used, as given in the Signetics manual.

R/C is the register or condition code, if one is required; either the symbols given earlier may be used, or a single-digit hexadecimal number.

SYM is a special symbol or symbols to indicate indirect addressing and/or indexing, if required.

OPND is the operand for the instruction; it may be a hexadecimal data number or an address, and if an adress it may be given either as a hex number or one of the labels for forward referencing. In the case of relative addressing, the assembler expects an absolute hex address, and will calculate the required displacement. The only exception is for ZBRR and ZBSR instructions, where the actual displacement must be typed in.

Each of the above symbol fields should normally be separated from those adjacent by one or more spaces. If the label field is not used, a leading space is not required although one or more spaces may be used if desired for appearance. The separator between the OPC and R/C fields may be a comma instead of a space, and the space between the SYM and OPND fields may be omitted if desired.

If the first character of a line is an asterisk (*), the assembler assumes the line is a comment only and ignores it. A comment line may have up to 56 characters apart from the asterisk.

The symbols used to indicate indirect addressing and indexing in the SYM field are as follows:

'*' Means indirect addressing.

'#' Means normal indexing. Note, however, that when indexing is specified the index register must be given in the R/C field, unlike the normal assembler format. This is no real problem since R0 is always the implied source/destination register for indexed instructions.

'+' Means indexing with auto-increment. Again the index register must be given in the R/C field.

'—' Means indexing with auto-decement. The index register must be given in the R/C field.

Where indirect addressing and indexing are to be specified in the one instruction, the two appropriate symbols are used together with the indirect addressing symbol given first. For example:

LODA,R3 *+8A0

which is a load indirect through address X'8A0, using R3 as the index register and with auto-increment. Thus R3 will be incremented and added to the address found in location 8A0 to generate the final effective address for the instruction.

The function of the label operators is to help you in writing forward memory references. That is, references in the operand field of instructions to locations in the program which have yet to be fed in, and are therefore not known in terms of their exact absolute address.

There are restrictions on the use of the label operators, as follows. They can only be used in the OPND field of branch instructions, and they cannot be used in relative addressing instructions. Nor can they be used with indirect addressing or indexing. This limits the use of the labels fairly severely, but they can still be quite handy.

Ten different label operators are allowed, represented by the symbols @0—@9. Each one can be used in the operand field of instructions any number of times before it is finally defined by specifying it in the label field of an instruction or pseudo-op. Note that all references to a label must precede its definition, due to the way in which the assembler handles the labels.

However after being defined a label operator may be re-used again.

As mentioned earlier, the assembler recognises four different directives, or pseudo-operators. These are basically instructions to the assembler itself, rather than symbolic instructions to be assembled into machine code. The four directives recognised are as follows:

ORG is a directive to the assembler to reset its program counter; i.e., the pointer which the assembler uses to store the assembled program instructions into memory. The format of this directive is

ORG nnnn

where 'nnnn' is a hexadecimal number specifying the new program counter value. Leading zeroes are not required.

ASCI is a directive to the assembler to store in memory a string of alphanumeric characters, in ASCII code. Following the directive mnemonic the assembler skips any leading spaces, then takes the next character it finds as a string delimiter. All of the following characters up to the next occurrence of the delimiter character are then stored as an ASCII string. The actual string may be up to 52 characters long. The format for this directive is thus

ASCI < delim >< string >< delim >

DATA is a directive to the assembler to store one or more data bytes in memory, beginning at the location given by the current value of the assembler's program counter. The directive format is

DATA nn nn nn nn nn nn nn ...

where each 'nn' is a two-digit hexadecimal number, and the numbers are separated by spaces. If an error is made while typing a number, it may be corrected merely by typing in the two correct digits before the terminating space. Leading zeroes are not required. Up to 18 data bytes may be entered on a line if no corrections are made.

END is the directive which is used to indicate to the assembler that no further source material is to be assembled. When this directive is encountered the assembler returns to PIPBUG.

If desired, comments may be added after the operand field on most source lines, providing the comments are separated from the operand by at least one space. The only type of source line where this cannot be done is one consisting of a DATA directive, as the assembler searches to the end of the source line for data numbers for this directive. No special symbol is required to distinguish comments following source instructions or directives.

The assembler resides in memory from location 15CC to 1AFE, inclusive. Part of this range is not used by the program itself, but is used as a line input buffer, scratchpad and label buffer area (1A02—1A54). The initial starting address is 1600, so after loading into memory the assembler is called by giving PIPBUG the command G1600r (where "r" is carriage return).

When called, the assembler first types out an identifying message: "2650 LINE ASSEMBLER". It then types out a suggested initial origin, which is X'0440 — the start of the available RAM above PIPBUG's scratchpad area. If you don't wish the assembled program to start at this address, you can immediately change the program counter to another value by using the ORG directive.

You can now type in your program to be assembled, line by line. When you conclude each line with the usual carriage return, the assembler will attempt to assemble it. If you have made no format (syntax) errors and it can do so, it will indicate this and its ability to accept a further line by typing the new value for its program counter at the start of the next line. You thus get a continuous indication that all is well, along with an indication of the memory space being used by your program.

If you make a format error and the assembler cannot assemble the line, it will abort and return to PIPBUG via the '?' error message routine. After working out what went wrong, you can return to continue the assembly by either re-starting at address 1600, or by starting at address 160E. The latter preserves any forward reference labels you may have been using, although the assembler's program counter is disturbed. You thus have to reset it with an ORG directive.

I have prepared a small demonstration of the assembler's use, which is shown in Fig. 1. As you can see the program assembled is a very short message printing routine which starts at X'0500, but its assembly illustrates most of the things you need to know about the assembler and the way it is used.

Note that the first three input lines are comments, which are effectively ignored by the assembler. Note also the way the assembler prints out the current value of its program counter at the start of each line, so that you can see how much memory the program is taking up. Needless to say you also make use of these addresses when typing in backward-referencing operands — an example of this is shown in the line commencing at address 050E.

Finally, note that after assembly, the program which had just been assembled was called from PIPBUG by typing G500. It then ran, typing out the simple message "HELLO THERE".

Needless to say, once you have assembled a program and checked that it runs, you can dump it in the normal way to cassette tape or paper tape using the normal PIPBUG dump routine.

Well, there it is — a small but very practical assembler which should make programming your 2650 very much easier. Incidentally for those who would like to analyse the assembler's operation in detail, full source listings will be available from our Information Service for a fee of $4.00, to cover photocopying and postage.